Datenbanken: Indexstrukturen

Stefan Halfpap, Ralf Teusner, Werner Sinzig, Michael Perscheid

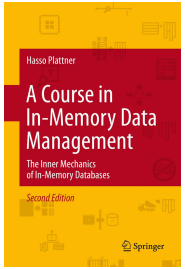
19. Juni 2020

Vorlesungsinhalte/-aufbau

- Einführung zu Unternehmensanwendungen
- Grundlagen des IT-gestützten Rechnungswesens und der Planung
- Einführung zu relationalen Datenbanken und Anfrageverarbeitung
- Grundlagen von (spaltenorientierten) Hauptspeicherdatenbanken
- **Trends in Hauptspeicherdatenbanken**
- Klausur

Literatur und Empfehlung

- Hasso Plattner „A Course in In-Memory Data Management”



- Andy Pavlo „Database Systems“ - Hash Tables & Tree Indexes

  https://15445.courses.cs.cmu.edu/fall2018/slides/06-hashtables.pdf

  https://15445.courses.cs.cmu.edu/fall2018/slides/07-trees1.pdf

  https://15445.courses.cs.cmu.edu/fall2018/slides/08-trees2.pdf

- Felix Naumann: Datenbanksysteme II Indexstrukturen (Klassische Indexstrukturen)

Indexstrukturen
Motivation

Anwendungen arbeiten nur mit einer kleinen Untermenge aller Daten

- Spaltenorientierung gut, da meist wenige Attribute verwendet werden
- Einige Anfragen beziehen sich nur auf wenige Tupel;
  Spaltenorientierung gut, da auf jeder Spalte effizient gefiltert werden kann

- Aber: Idealer Weise können Tupel direkt gefunden werden ohne alle Zeilen der Tabelle zu durchsuchen
- Zusätzlich lassen sich komplexe DB-Operationen, z.B. Join und Aggregation, mit Indexstrukturen effizient umsetzen

Idee: Nutzung von Hilfsstrukturen (Effizienz vs. Speicherverbrauch)

Indexstrukturen
Keine Struktur ist die Beste (für alle Anwendungen)

- Zugriffstypen: Gleichheit vs. Bereichsabfragen
- Zugriffszeit – Zeit ein Datenelement /eine Menge von Elementen zu finden
- Einfügezeit – Such-/Zugriffszeit + Aktualisierung der Indexstruktur
- Löschzeit – Such-/Zugriffszeit + Aktualisierung der Indexstruktur
- Speicherverbrauch

Indexstrukturen
Definition

- Hilfsstruktur, die Zugriff auf Tupel beschleunigen
  - Suchschlüssel: ein oder mehrere Attribute
    (Partieller Index: Untermenge der Attribut-Werte) https://www.postgresql.org/docs/current/indexes-partial.html
    (Covering Index: Zusätzliche Attribute neben Suchschlüssel) https://www.postgresql.org/docs/current/indexes-index-only-scans.html
  - Nachteile: Speicherverbrauch und Aktualisierungskosten
- Klassifikation
  - Sortierte Indexe: Basieren auf sortierte Reihenfolge der Schlüsselattribute
  - Hash-Indexe: Basieren auf (Hash-)Funktion, die Schlüssel gleichmäßig auf Behälter aufteilt

Indexstrukturen
Agenda

- **Bedeutung von Indexstrukturen für spaltenbasierte DBs**
- **Indexstrukturen für Dictionary-kodierte Spalten**
- Indexstrukturen
  - Überblick
  - Kuckucks-Hashing
  - Skip-Liste
  - Bloomfilter
  - Gelernte Indexstrukturen

Kurzvorstellung von Datenstrukturen/Konzepte, die über "Standard-DB-Vorlesungen" hinausgehen

https://arxiv.org/abs/1712.01208

Kraska et al.: The Case for Learned Index Structures. (2017)

Indexstrukturen
Bedeutung

- Anwendungen arbeiten nur mit einer kleinen Untermenge aller Daten
  - Spaltenorientierung gut, da meist nur 10% der Attribute verwendet werden

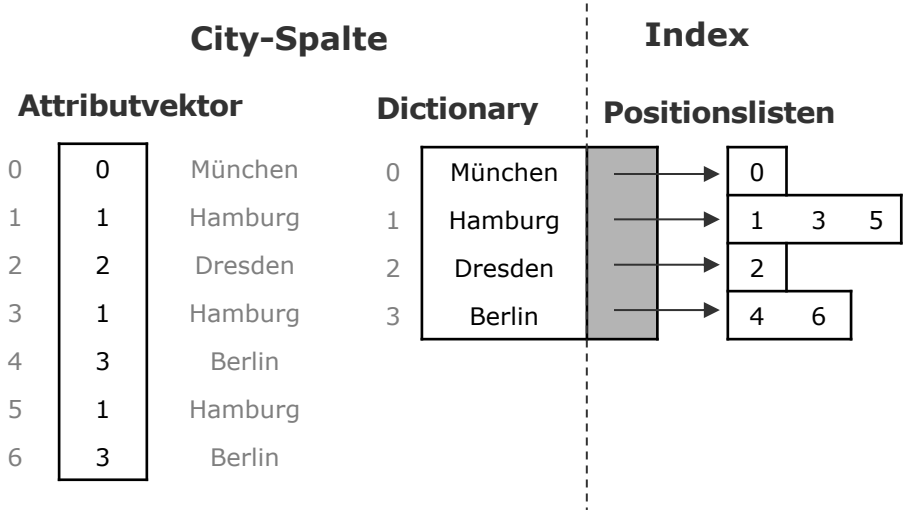
    Hasso Plattner: A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database. (2009)

  - Scan auf jeden Spalte effizient um Tupel zu filtern
    (Indexstrukturen bei zeilenorientieren wichtiger)
- Aber: Komplexität des Scans ist O(n)
  - Indexstrukturen können insbesondere selektive Scans (mit kleiner Ergebnismenge) auf Tabellen mit vielen Tupeln beschleunigen

    Kester et al.: Access Path Selection in Main-Memory OptimizedData Systems: Should I Scan or Should I Probe? (2017)
    https://stratos.seas.harvard.edu/files/stratos/files/accespathselection.pdf
  - Schlüsselbedingungen beim INSERT überprüfen
  - Unsortiertes Dictionary benötigt Index um effizient Wert-IDs für Werte zu finden
- Temporäre Indexstrukturen zur Anfragebearbeitung, z.B. beim Hash-Join

Indexstrukturen für Dictionary-kodierte Spalten

- Dictionary speichert bereits alle verschiedenen Werte → Schlüsselattribute des Index
  - Idee: speichere pro Schlüsselattribut eine Positionsliste die alle Offsets in den Attributvektor pro Attributwert speichert
    + Baum um Werte im Dictionary zu finden (siehe 07_DB_Weitere_Optimierungen)



Indexstrukturen für Dictionary-kodierte Spalten

Sortiertes Dictionary (Main Partition oder komprimierter Chunk)

Faust: Fast Lookups for In-Memory Column Stores:Group-Key Indices, Lookup and Maintenance. (2012)

- Partition/Chunk ist unveränderbar (bis auf Invalidierungen)
- Index muss nicht aktualisiert werden
- Attributvektor ordnet (impliziten) Positionen Werte (Wert-IDs) zu
- (Invertierter) Index ordnet Werten Positionen zu

City-Spalte

Attributvektor (AV)

0	4	Hannover
1	2	Dresden
2	3	Frankfurt
3	2	Dresden
4	1	Berlin
5	0	Aachen
6	1	Berlin
7	5	Iserlohn

Sortiertes Dictionary

0	Aachen
1	Berlin
2	Dresden
3	Frankfurt
4	Hannover
5	Iserlohn

(Invertierter) Index

Indexoffsets
Offsets im IP

0	0
1	1
2	3
3	5
4	6
5	7

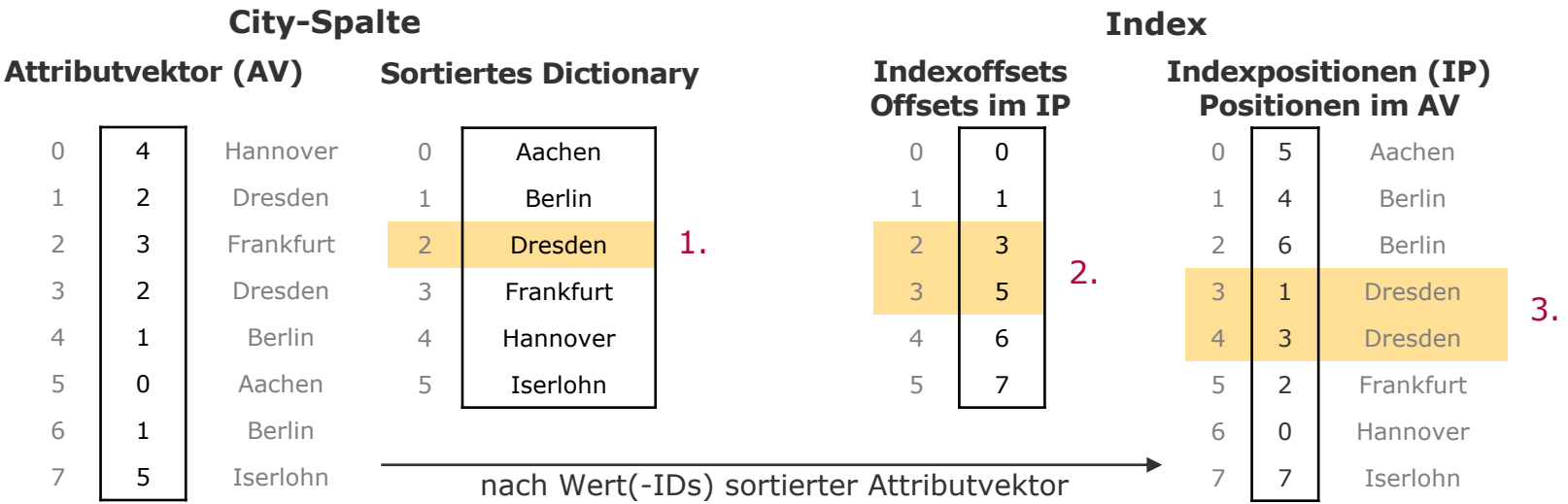
Indexpositionen (IP)
Positionen im AV

0	5	Aachen
1	4	Berlin
2	6	Berlin
3	1	Dresden
4	3	Dresden
5	2	Frankfurt
6	0	Hannover
7	7	Iserlohn

nach Wert(-IDs) sortierter Attributvektor

Indexstrukturen für Dictionary-kodierte Spalten
Filter Spalte mit sortiertem Dictionary nach ‘Dresden‘

1. Suche Wert-ID für ‘Dresden‘ im sortieren Dictionary → 2
2. Lese Indexoffsets für gefundene Wert-ID 2 → [3; 5)
3. Lese Positionen zwischen den Offsets [3; 5) → [1, 3]



Indexstrukturen für Dictionary-kodierte Spalten
Unsortiertes Dictionary (Delta Partition)

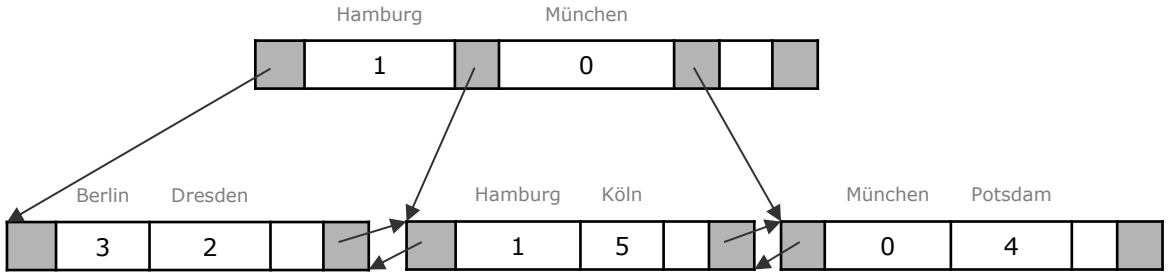
Karnagel et al.: Improving In-Memory Database Index Performance with Intel Transactional Synchronization Extensions. (2014)

- Unsortiertes Dictionary benötigt Index um effizient Wert-IDs für Werte zu finden
- Beispiel: INSERT von ‘Potsdam‘ in den Attributvektor benötigt Wert-ID für ‘Potsdam‘

City-Spalte

	Attributvektor (AV)			Unsortiertes Dictionary
0	0	München	0	München
1	1	Hamburg	1	Hamburg
2	2	Dresden	2	Dresden
3	1	Hamburg	3	Berlin
4	3	Berlin	4	Potsdam
5	4	Potsdam	5	Köln
6	5	Köln		

Index für unsortiertes Dictionary



Indexstrukturen
Überblick

- Baum ist wichtigste Indexstruktur für Datenbanken
  - Effiziente Indexerstellung (mit dynamischer Größe)
  - Unterstützt Bereichsabfragen und sortiertes Lesen
  - Effiziente Nebenläufigkeitskontrolle

Viele Varianten und Optimierungen ... Goetz Graefe: Modern B-Tree Techniques. (2011)

- Hash-Index als Alternative mit ebenfalls vielen Varianten - hier: **Kuckucks-Hashing**

- **Skip-Liste** als weitere Indexalternative
- **Bloomfilter** als Struktur um unnötige Zugriffe auf Partitionen zu vermeiden
  „Partition Pruning“ (siehe 07_DB_Weitere_Optimierungen)

Indexstrukturen
Hash-Index - Kuckucks-Hashing

- Problem beim Hash-Index - Kollisionen
- Idee: nutze mehrere Hash-Funktionen/-Tabellen
  - Überprüfe beim Einfügen alle Hash-Tabellen und wähle eine mit einem freien Platz
  - Falls keine Tabelle einen freien Platz hat: vertreibe das Element von einer Tabelle und suche für dieses einen Platz in einer anderen Tabelle

    (Erstelle die Tabellen neu, wenn ein Zyklus (Endlosschleife) entsteht)

→ Suche und Löschen sind immer O(1),
weil nur eine Position pro Hash-Tabelle überprüft werden muss

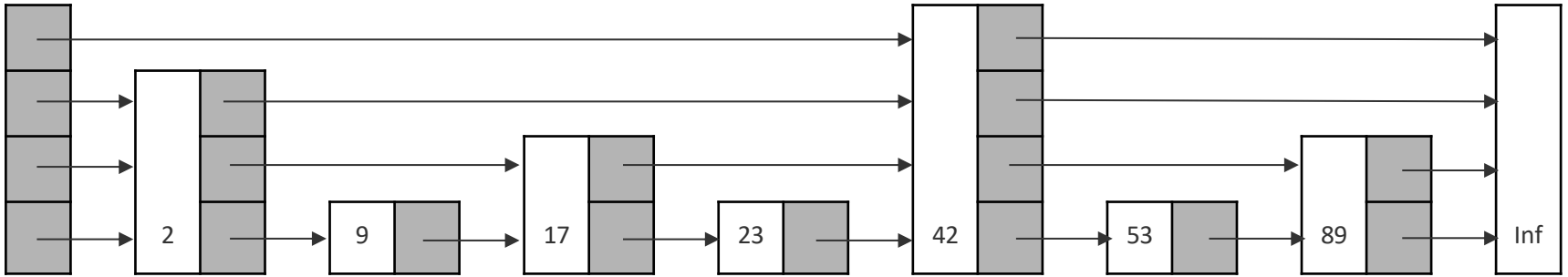
http://www.cs.tau.ac.il/~shanir/advanced-seminar-data-structures-2007/bib/pagh01cuckoo.pdf

Indexstrukturen
Skip-Liste

William Pugh. Skip lists: **a probabilistic alternative to balanced trees**. (1990)

- Einfache Variante einer dynamischen Datenstruktur, die Sortierung bewahrt ist, ist eine sortierte verkettete Liste
- Aber **lineare** Suche

- Idee: Verwende zusätzliche Pointer um über Listenbereiche zu springen um effizienter zu suchen (Speichere zusätzliche Pointer um binäre Suche zu imitieren)



Pruning Filter
Bloomfilter

- Bisher: Effizientes Suchen von vorhandenen Werten

- Häufig hilft es auch zu Wissen, dass Werte nicht vorkommen,
  z.B. zum Partition Pruning, das ist die Vermeidung unnötiger Scans von Partitionen, wenn keine relevanten Daten in der Partition gespeichert sind
  - Indexstrukturen können dazu auch verwendet werden, benötigen in der Regel viel Platz
  - Idee: Verwende Speicher-effiziente probabilistische Strukturen z.B. min/max-Aggregate oder **Bloomfilter**

Pruning Filter
Bloomfilter

Bloomfilter: Verwende eine kleine Bitmap um alle gespeicherten Werte zu repräsentieren

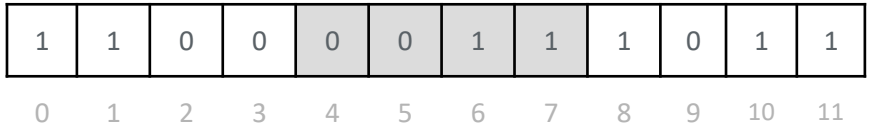
- Leere Partition entspricht Bitmap mit nur Nullen
- Einfügen: Jeder gespeicherte Wert erzeugt ein Bitmuster, der sich auf die Bitmap legt: Bitmuster wird durch eine Menge von Hash-Funktionen bestimmt
- Suchen: Bitmuster für gesuchten Wert mit Bitmap abgleichen:
  - Wert ist mit Sicherheit nicht vorhanden, wenn ein gesetztes Bit des Bitmusters in der Bitmap Null ist
  - Ansonsten: ist der Wert *wahrscheinlich* vorhanden
- Löschen: beim normalen Bloomfilter nicht unterstützt (Zählende Bloomfilter als Erweiterung)

Pruning Filter
Bloomfilter - Beispiel

Gespeicherte Werte

key	Hash1(key)	Hash2(key)	Hash3(key)	Bitmuster
Hannover	0	6	1	1100 0010 0000
Berlin	7	10	8	0000 0001 1010
Köln	6	0	8	1000 0010 1000
Hamburg	0	11	0	1000 0000 0001

Bloomfilter



- Suche ‘Berlin’: 0000 0001 1010
- Suche ‘Potsdam’: 0001 0100 0000
- Suche ‘Bielefeld’: 0100 0010 1000 ← Falsch positive (Wert ist nicht vorhanden, aber mit Bloomfilter nicht feststellbar)

  Wahrscheinlichkeit bei 4 Werten, 12 Bits und 3 Funktionen: 0.253

  http://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html

Zusammenfassung

- Indexstrukturen sind für effiziente Scans in spaltenbasierten DBs nicht so wichtig wie in zeilenorientierten
- Dennoch sind Indexstrukturen für Überprüfung von Schlüsselbedingungen, INSERTS bei Spalten mit unsortiertem Dictionary, häufigen selektiven Scans und zur effizienten Anfragebearbeitung auch für spaltenbasierte DBs wichtig
- Bäume und Hash-Tabellen sind die wichtigen Indexstrukturen für DBs
- Kuckucks-Hashing verwendete mehrere Hash-Funktionen/-Tabellen für Kollisionen
- Skip-Listen sind Erweiterung von verketteter Liste mit effizienter Suche
- Bloomfilter ist eine probabilistische Datenstruktur, z.B. zum Partition Pruning